# 5
# Structured programming

*If you can't solve a problem, then there is an easier problem you can solve: find it.*
George Polya, mathematician

## Overview

In the previous two units we have seen how to set up algorithms and how to convert them into programs. Some of the computer tasks we have to face though may not be straight forward enough to solve with a simple algorithm or program.

In this unit we will look at a more formal way of going about structuring a solution to complex programming tasks. Before we do that however we will investigate some of the more useful ways of handling data.

In this unit we will look at:

- arrays
- sorting
- records
- string handling
- file handling
- malware
- the software development cycle
- structured programming.

## Arrays

In the programs and pseudocode we have seen previously we have worked with individual variables. To enter three values we have written steps such as:

```
first =18
second = 34
third = 12
```

© Kevin Savage 2011 – Licensed to St Mary's Catholic College for use in 2012

But what would happen if we wished to enter twenty values, or fifty, or say we were writing a program to sort 5 000 scores into order? Obviously it is going to be impractical to have a separate variable name for each of 5 000 pieces of data. The answer is to use arrays.

An *array* is a series of memory spaces with the same name, similar to sub-scripted variable names you might use in Maths or Physics such as $x_1$, $x_2$, $x_3$. We distinguish each element of the array by the number index that follows it. For example: *score(1), score(2), score(3), score(4)* is an array called *score* that can hold four different values.

## Array input and output

To enter a value into an array we use an assignment statement. Written in pseudocode they would look like:

```
year(1) = 2005
year(2) = 2006
year(3) = 2007
```

Output is similar:

```
write year(1), year(2), year(3)
```

Simple input and output statements are fine for just a few elements in an array, but to save writing the same input or output statements over and over we can use counted *do* or *for* loops such as the following:

```
do j = 1 to 50
    prompt for year(j)
    get year(j)
enddo
```

This would loop 50 times, each time getting the user to enter a value that would be stored in *year(1), year(2),* etc., up to *year(50).*

Output could be done similarly:

```
do j = 1 to 50
    write year(j)
enddo
```

In both cases the loop counter *j* becomes the index value of the array variable each time the loop iterates.

It is traditional in programming to use loop counters with the letter names of *j, k, l,* and so on, when working with arrays. These counter variables are almost always an ordinal (counting) type such as integer (1, 2, 3, ....) or char (‘a’, ‘b’, ‘c’,...).

As a further example, say we wished to record test scores for each of 25 students. To do this we could use the following algorithm in pseudocode to load the array:

```
do j = 1 to 25
      prompt for score(j)
      get score(j)
enddo
```

We can now do various operations with this array such as get an average:

© Kevin Savage 2011 – Licensed to St Mary's Catholic College for use in 2012

```
total = 0
do k = 1 to 25
      total = total +  score(k)
enddo
average = total/25
write “The average mark for the test is ”, average
```

or we could find the highest and lowest scores:

```
max = 0
min = 1000
do l = 1 to 25
      if score(l) > max then
          max = score(l)
      endif
      if score(l) < min then
          min = score(l)
      endif
enddo
write “The highest score is ”, max, “ and the lowest score is ”, min
```

In this latter algorithm each element in the array is compared with the two starter values of 0 and 1 000. On the first run through of the loop *score(1)* will be both the highest (>0) and lowest (<1 000) score and so its value will be stored in *max* and *min*. After this, in turn, each other element in the array is compared to the current values in *max* and *min*, with *max* and *min* being updated as new high or low scores come up.

## Multidimensional arrays

Arrays are not limited to one dimension but can be two (or more) dimensions. A 2D array has cells similar to the (x, y) coordinates of a graph. These are useful to store data that can be represented in table form such as the following:

| Maths | English | Science | ... |
|---|---|---|---|
| 12 | 23 | 11 | ... |
| 7 | 9 | 12 | ... |
| 18 | 19 | 13 | ... |
| : | | | |

To store the marks for 20 students for each of 5 tests we would use nested loops:

```
do j = 1 to 20
      do k = 1 to 5
          prompt for mark
          get mark(j, k)
      enddo
enddo
```

© Kevin Savage 2011 – Licensed to St Mary's Catholic College for use in 2012

Each time the outer loop ran it would cause the inner loop to iterate five times. This would store *mark(1, 1), mark(1, 2), mark(1, 3),* and so on, up to *mark(20, 5).*

We are not limited to two dimensions. If we wish we can have arrays with three or more parts to each element. Later we will look at specialised arrays called records that can be used to store more complex data.

### Declaring arrays

In most 3GLs before an array can be used in a program it must be declared, the form of declaration depending on the language to be used.

The declaration identifies the name of the variable, if it is one or multi-dimensional, the type of variable it involves, and may indicate the range of values the array can hold.

e.g.

| | |
|---|---|
| C, C++: | `int score[25];` |
| | `int mark[20][5];` |
| Pascal: | `score : array[1..25] of integer;` |
| | `mark : array[1..20, 1..5] of integer;` |
| Java: | `int[ ] score;` |
| | `int[ ] [ ] mark;` |
| Visual Basic: | `Dim Score(1 To 25) As Integer` |
| | `Dim Mark(1 To 20, 1 To 5) As Integer` |

In some languages, such as PHP, there is no requirement to declare arrays. The first entry into the array sets up its structure.

e.g.

```
$score = Array("65");
$mark = Array(Array("6", "A"));
```

Any further additions to the array are then just added as needed.

As you do more and more programming you will find that using arrays is a very efficient way of handling multiple values, as we shall see in the next section on sorting.

## Activity 5.1 – Collections

1. We can have arrays of any type of variable (*integer*, *string*, *char*, etc.).

   Write the declaration to be used for each of the following in the 3GL of your choice:

   a an array to hold the ages of 15 people

   b an array to hold 10 colours

   c an array to hold the cost in dollars and cents of 1000 items

   d an array to hold 200 addresses

   e an array for 30 people's gender (M/F)

   f a 2D array to store 5 event names with up to 8 students entered for each event.

© Kevin Savage 2011 – Licensed to St Mary's Catholic College for use in 2012

2. Suppose an array has the following elements holding the values indicated:

| Element: | time(1) | time(2) | time(3) | time(4) | time(5) |
|---|---|---|---|---|---|
| Value: | 15 | 20 | 12 | 23 | 18 |

What is held in *answer* as a result of each of the following:

a
```
answer := time(3);
```

b
```
answer := time(1+4);
```

c
```
number = 2
answer = time(number)
```

d
```
answer = time(2) + time(3)
```

e
```
answer = time(4) * time(1)
```

3. What values will be loaded into the following arrays (record your answer as demonstrated in Q2 above):

a
```
do j = 1 to 5
    number(j) = j * 2
enddo
```

b
```
do k = 1 to 6
    cell(k) = k * k
enddo
```

c
```
count = 1
do l = 1 to 8
    count = count * 2
    place(l) = count
enddo
```

d
```
do j = 1 to 4
    do k = 1 to 4
        write j, " times ", k, " is ", j * k
    enddo
enddo
```

4. Write a program to enter ten numbers into an array.

   The program must then pick the largest and the smallest number, and calculate the average of the numbers.

5. Prepare a program that will record the cricket scores of the *Bunyip Wanderers*. The program will use an array to store the scores of the 11 cricketers in the team.

   The scores will be entered by the user into the array, totalled, and an average calculated. The program will display the total score, the team average, and also indicate the highest and lowest score of the team.

   Include effective internal documentation.

© Kevin Savage 2011 – Licensed to St Mary's Catholic College for use in 2012

## Bubble sort

In order to explore how arrays can be used we will look at the bubble sort.

Computers handle huge volumes of data. When humans come to look through this data we need it arranged in some way so that we can find what we are looking for. Often we require the computer to put the data into alphabetical order, or to organise numbers in order with the largest first.

To reorder the data by computer we use *sorting algorithms*. There are many of these because it is such an important part of computing, but for now we will only look at one.

A *bubble sort* involves moving down a list and comparing each pair of values. If necessary the values are swapped. The process returns to the beginning of the list again and again until all of the numbers are in the correct order. The larger numbers rise to the top of the list like bubbles in water. This method is boring and repetitive for humans but (you guessed it) effective for computers.

Here is the bubble sort algorithm (not in pseudocode):

```
repeat until list is in order
    if at end of list then
        return to beginning
    else
        compare next two values
        if second larger then
            swap values
```

Let's look at a bubble sort in action on the following set of values, 17 23 14 28 15
The list is not in order and we are not at the end so compare 17 and 23

| | |
|---|---|
| the 23 is larger so swap them | 23 17 14 28 15 |
| compare 17 and 14; 14 is not larger so we leave them | 23 17 14 28 15 |
| compare 14 and 28; the 28 is larger so swap them | 23 17 28 14 15 |
| compare 14 and 15; the 15 is larger so swap them | 23 17 28 15 14 |

We have now made one run through of the list. At this stage it is still not in order so we start a second run through.

Compare the first two values:

| | |
|---|---|
| 17 is smaller than 23 so leave them and continue | 23 17 28 15 14 |
| 28 is larger than the 17 so swap them | 23 28 17 15 14 |
| the 17 and 15 are in correct order | 23 28 17 15 14 |
| as are the 15 and 14 | 23 28 17 15 14 |

The list is still not in order so we start a third run through:

| | |
|---|---|
| 23 is smaller than 28 so swap them and continue | 28 23 17 15 14 |
| 23 and 17, 17 and 15 and 15 and 14 are in order. | 28 23 17 15 14 |

The list is now in descending order of size.

© Kevin Savage 2011 – Licensed to St Mary's Catholic College for use in 2012

Note how values like 28 “bubble” up closer to the top of the list during each run through. This same algorithm will work as well on 10 000 numbers, or on other data such as names.

## Bubble sort algorithms

The only effective way to code a bubble sort algorithm is to use an array to store the values. This makes it easy to compare pairs of values and, if necessary, swap them over.

The values to be sorted can be input, and the sorted values can be output as shown in the last section. To input, and later output, 10 values we can use the following:

```
do j = 1 to 10
    prompt for value
    get value(j)
enddo
```

```
do l = 1 to 10
   write value(l)
enddo
```

The swapping of values is performed by the following algorithm:

```
do k = 1 to 9
    if value(k) < value(k+1) then
        temp = value(k)
        value(k) = value(k+1)
        value(k+1) = temp
    endif
enddo
```

*Temp* is a temporary or holding variable. It stores the first value while the second overwrites it. It is then used to overwrite the second.

At this stage we can input the unsorted numbers, we can swap values over, and we can output the sorted list. The next step is to go through the list repeatedly until it is in order. The simple way to do this is to run it nine or ten times so that all pairs of numbers are compared. There is however a better, more efficient way.

What we need to have is some way of recording if a swap has occurred. If on a run through of the list there has been no swap made then the list must be in order, and there is no need to keep looping. To do this we are going to use a *boolean* variable (true/false) that will act as a *flag*. This will be set to *false*, and will only be made *true* if a swap occurs. If after a run through the flag stays false then there has been no swap made and we can stop. We will call the boolean *cont* (for continue):

```
cont = true
while cont = true
    cont = false
    do k = 1 to 9
        if value(k) < value(k+1) then
            temp = value(k)
            value(k) = value(k+1)
            value(k+1) = temp
            cont = true
        endif
    enddo
endwhile
```

© Kevin Savage 2011 – Licensed to St Mary's Catholic College for use in 2012

The *while* loop will only iterate as long as *cont* is true. As soon as the loop begins *cont* is made *false*, and it will only be made *true* if any swap is made. When there are no more swaps the *while* loop finishes and the program moves on to display the list in order.

(Why is *cont* made *true* before the *while* loop starts?)

## Records

So far we have seen how arrays can store this data, but generally arrays are limited to a single data type of integer, or string, or some other type.

Many 3GLs have a more powerful data structure known as a *record* type. A record is like a multi-dimensional variable, but one in which different types can be stored under one label.

Usually the record type is declared separately, for example in Pascal the following declaration to record student information:

```
name = record
    firstName : string[12];
    surname : string[20];
    idNumb : integer;
    average : real;
    end;
```

This creates a record type called *name* that has four fields, the first name, surname, identity number and subject average for a student. Note that the different parts of the record are of different variable types (*string*, *integer*, *real*).

Once the record type has been made then an array of the record can be declared; again in Pascal:

```
var student : array[1..50] of name;
```

This array will hold up to 50 sets of information, all of which have the structure identified in *name* above.

Individual elements of the record are accessed through a dot notation such as:

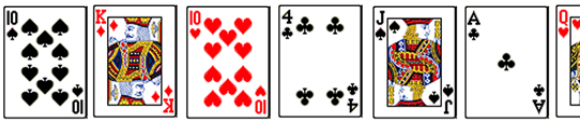
```
student[j].firstName    or    student[j].average
```

Records are loaded and displayed the same way as normal arrays.

# Activity 5.2 – Getting sorted

1. Line up 10 unordered playing cards.

   Follow the steps of the bubble sort algorithm to rearrange the cards in order of size.



2. a Convert the bubble sort algorithm above into pseudocode.

   b From your pseudocode prepare a program in a 3GL to sort 10 numbers into order.

   Test and run it to check it works for a range of input values.

   c If your 3GL has the facility to watch or trace values set it up to observe the values in the array as you step through the program.

   d Adjust your program so that it can work with any number of input values.

© Kevin Savage 2011 – Licensed to St Mary's Catholic College for use in 2012

3. The bubble sort algorithm is little used nowadays.
   a Find out why the bubble sort is not used any more.
   b Identify alternative sorting algorithms in use.
   c Choose one of these and prepare a program to run it.
4. Prepare a record declaration in a 3GL of your choice to store information about a CD collection.

   The record is to have fields for title, artist, year, and number of tracks.

## String handling

Most of the algorithms and programs we have investigated so far have involved calculations with numbers. The majority of 3GLs however also have a range of string or char handling functions that can work with or manipulate text.

Text is composed of a list or string of characters. For example the following string:

```
39 Middleford Rd, Maleny
```

holds 24 characters. The numbers *3* and *9*, the spaces, and the comma all count as characters. These characters are usually recorded in memory using the ASCII code.

It is usual to enclose strings in quotes, i.e. “ ” or ‘ ’, depending on the language. These indicate clearly the beginning and end of the string, especially if there are leading or trailing spaces.

Text is loaded into a string with an assignment statement and output as normal.

```
address = “39 Middleford Rd, Maleny”
write “The address is ”, address
```

The output statement above has two strings. The one in quotes is called a literal, while the second is stored in the variable *address*.

The principle string handling functions available include the following

- search – look through text to find specific characters, or the position within the text of specified characters
- copy – reproduce some or all of a set of characters in text; a smaller string extracted from a larger one is called a *substring*
- insert or append – add specific characters into text, or add them at either the beginning or end
- replace - change specific characters in text with other characters
- concatenate – join two pieces of text into one
- delete or trim – remove specific characters from text, or remove them from either the beginning or end
- compare – check if one piece of text is identical to another, or if its length is the same, or more or less than another
- order – place strings in alphabetical (or other) sequence
- count or length – find the number of characters in text
- convert or replace – change some or all of the text into a different form, e.g. lower to upper case

© Kevin Savage 2011 – Licensed to St Mary's Catholic College for use in 2012

- encryption or decryption – converting the text into, or from, a more secure form

Most programming languages store text as an array in memory, but as long as the text is declared as a string or char we can handle it like an ordinary variable.

## Activity 5.3 – Stringing along

1. Use the following strings to answer the questions below:

   ```
   first = 'nevertheless'
   second = 'fortunately'
   third = 'Now is the time'
   ```

   a How many characters in *second*?

   b How many characters in *third*?

   c What 3 character substring is extracted from *first* starting at character 6?

   d What 4 character substring is extracted from *first* starting at character 2?

   e Insert 'not ' into *third* starting at character 8.

   f Delete 2 characters from *second* starting at character 10.

   g Convert *first* to uppercase.

2. Working with the examples above use string handling functions in a 3GL of your choice to:

   a Find the how many characters are in *first.*

   b Find the how many characters are in *second.*

   c Extract the word *ate* from *second*.

   d Insert the word *only* into *third* between the third and fourth words.

   e Delete the third word from *third*.

   f Change *second* to read "fortune".

   g Join *second* to the end of *third*.

3. Prepare a program to get the user to enter a person's first name, then their middle initial, and then surname. After this use concatenation to join the three into one string. Be careful to include spaces.

4. Place the following words in ASCII order, "smallest" first:

   cat catch can candle call Cat cane candy case Car

5. Look at the following pseudocode:

   ```
   if grade = "junior" then
       rate = 15.5
   else
       rate = 18.75
   endif
   ```

   a Explain what the algorithm does.

   b Where do you think it could be used?

© Kevin Savage 2011 – Licensed to St Mary's Catholic College for use in 2012

6. A palindrome is a word or phrase that reads the same forwards as backwards. Examples include *madam*, *Glenelg*, *Madam in Eden I'm Adam*, and one supposedly attributed to Napoleon, *Able was I ere I saw Elba*.

   The following algorithm will detect if a text string is a palindrome or not:

```
flag = true
prompt for sample
get sample
count = length(sample)
do j = 1 to (count/2)
    start = copy(sample, j, 1)
    finish = copy(sample, count – j + 1, 1)
    if start < > finish then
          flag = false
    endif
enddo
if flag = false then
    write "This is not a palindrome"
else
    write "This is a palindrome"
endif
```

   The *length* function finds out how many characters there are in the sample while the *copy* function will find extract a substring 1 character long from *sample* starting at the position indicated by the middle parameter.

   By working its way in from either end the algorithm finds if pairs of letters are the same. If any are not then *flag* is set to false. This works with both even and odd length strings (why?).

   a Explain how the algorithm works.

   b Code the algorithm in a 3GL of your choice.

7. Prepare a program that counts the number of words in an input text sentence. Do this by counting how many spaces there are and adding 1 if there is a full stop at the end.

8. Write a program that will encode a message the user enters.

   You may use any code you wish, but the Caesar cipher is one possible method. In this each character in the message is replaced by a letter say 16 further on in the alphabet.

   Extend the program so that a message that has been encoded by the program, can be decoded by the program.

© Kevin Savage 2011 – Licensed to St Mary's Catholic College for use in 2012

## File handling

So far we have been limited to the data that can exist in the computer's memory. When the computer is turned off the data is lost. To save things more permanently it is necessary to save the data to disc.

There are three classes of disc file:

- sequential or *text* files for simple read / write operations to get sections of text onto and off from a disc
- random or *typed* files to work with the user defined record type structures we investigated above
- binary or *untyped* files that are similar to *typed* files but with less structure so that the programmer has more control over precisely how data is written to, and read from disc.

In each case there is some way of indicating where the file begins and ends on the disc so that it can be told apart from other data on the disc.

To read data from disc most 3GLs have an *eof* (end of file) function that can be used with a *while* construct. Steps within the *while* loop will continue until an end of file marker is read.

The following algorithm uses the pseudocode keyword *read* to indicate data is read from disc into the variables indicated.

```
read ID, name, result
while not eof
    case of result
        1 to 30 : grade = "E"
        31 to 50 : grade = "D"
        51 to 65 : grade = "C"
        66 to 80 : grade = "B"
        81 to 100 : grade = "A"
    endcase
    write ID, name, result, grade
    read ID, name, result
endwhile
```

Three values are read from disc into the variables *ID*, *name* and *result*. The *while* loop then runs until the *eof* marker is detected. Each time it iterates a value is assigned to *grade* in the case statement, the values in the variables are output to screen, and then the next three values are read from disc.

Depending on the 3GL used it may also be necessary to open a file before reading from it, close it after use, and reset it if using it again.

## Activity 5.4 – Disc crazy

1. Sportz-Craz have a variety of items in stock (cricket bats, netballs, trampolines, etc.). Each item has an ID code and a cost price (e.g. Argus Cricket Bat 1876 $87.50).

   Plan, write and run a program to save onto disc a least five pieces of equipment in a sequential file. Call the file *sportz.dat*.

© Kevin Savage 2011 – Licensed to St Mary's Catholic College for use in 2012

2. Once we have the data on disc we can now access it to determine some information about it. Use the following algorithms to prepare procedures to query the data in *sportz.dat* (use a separate subroutine for each):

   a Stock count

```
open Sportz
count = 0
while not eof (Sportz)
    read item, code, cost
    count = count + 1
endwhile
stock = count/3
write stock, " items in stock"
close Sportz
```

   b Value of stock

```
open Sportz
total = 0
while not eof (Sportz)
    read item, code, cost
    total = total + cost
endwhile
write "The total value of stock is $", total
close Sportz
```

   c Cheapest item in stock

```
reset Sportz
min = 1000
while not eof (Sportz)
    read item, code, cost
    if cost < min then
        min = cost
    endif
endwhile
write "The cheapest item in stock costs $", min
close Sportz
```

3. a Create a file called *speeding.dat* to record information about fines registered by a police speed camera. The file will contain information such as date, vehicle registration number, speed, and fine owing. Populate the file with at least six sets of information.

   b Develop a subroutine that can query the file by prompting the user to enter the vehicle registration number and display all data in that record.

   c Add another subroutine that will allow the user to enter a date and will then display all speeding offences for that day and a total of the fines owing on that date.

   d Add a subroutine to update the file by adding more records to it. (Read all data from disc into a record array. Add to the array, and then write the extended array to disc.)

   e Add a fourth subroutine that will delete a record when a fine is paid.

© Kevin Savage 2011 – Licensed to St Mary's Catholic College for use in 2012

# SEI 5 – Malware

Not everyone who writes a computer program does it for ethical reasons. There are those who use their programming skills to develop a category of programs that are collectively known as malicious software, or *malware*. Malware is any program designed to alter the way a computer or information system works without the owner's knowledge or consent.

For many years the computer virus was the only form of malware. While there are many other variants of harmful code nowadays, the generic term of virus is often used (incorrectly) to label them all.

## Forms of malware

The original form of malware was the computer *virus*. These are infected programs that act as carriers similar to humans infected with a biological virus. Viruses copy themselves from computer to computer either over networks or through portable media such as flash drives or external HDDs.

The virus is a program that relies on the host computer to execute its code. When activated the virus delivers a *payload* of some kind. This payload may be as benign as displaying a message, or as harmful as deleting or altering files. A specialised form of virus is a computer *worm*. A worm usually consists of some form of script that is spread from computer to computer. It differs from a true virus in that it spreads across a network or the internet, and not through infected media such as discs.

A *rootkit* is malware that allows an outside user access to a computer without the owner of the computer being aware of it. This access can then be used to control the computer or to run *keyloggers* to gather personal details, passwords, credit card numbers, or other financial information. Similar to a rootkit is a *backdoor*, code that allows unauthorised users to bypass the normal authentication process to a computer or system.

*Spyware* is a program designed to gather information about users or alter the actions of the computer it is on. Spyware is installed secretly and attempts to remain hidden so that users are not aware it is there. The installation is carried out by stealth either by piggybacking on the installation of some other software, or by tricking the user into installing it by pretending it does something else. When in place the spyware can collect personal information, monitor user actions, redirect web browsing, or even install additional malware such as a rootkit. A specialised form of spyware is *adware* – software that is designed to monitor user preferences and then download and display advertisements targeted at that user.

*Trojans* or *Trojan horses* are a category of malware. These are programs that masquerade as something else. One example is a supposedly helpful program that the user downloads, that in reality installs spyware or adware. Another is an email attachment that rather than playing a video or performing a service carries out an unwelcome function in the system. There are also some pop-up windows in browsers that act as Trojans when clicked. As a general rule never open an email attachment, click on a pop-up ad, or run any executable program that you are not positive is safe.

A *botnet* is a collection of computers that have been compromised into running programs the owners did not authorise. This may be as a result of actions of viruses, Trojans, spyware, or the installation of a rootkit. When such compromised computers are used remotely they are called

© Kevin Savage 2011 – Licensed to St Mary's Catholic College for use in 2012

*zombie* computers; the person who initiates the botnet is known as a *bot herder* or *bot master*. Zombies can be used to send spam or to take part in distributed denial-of-service attacks. The advantage to the bot master is that not only is he or she using the collective power of many computers, but they are remote from the offending machines and so cannot be directly linked to the offence.

## Protection from malware

There are probably four classes of people who author malware:

- those who write malware in response to the intellectual challenge involved, to see if they can do it, or as a prank; many are show-offs who want to brag about their skill or to demonstrate their ingenuity at discovering a security loophole
- vandals whose aim is to create as much disruption as possible, taking delight in knowing they cannot be caught; their actions may be motivated by feelings of powerlessness, a desire for revenge, or simply the thrill of destruction
- those who aim to trick people into releasing information or carrying out activities they otherwise would not do; these enjoy the feeling of being able to manipulate others; this is a process known as *social engineering*
- those who seek financial gain through spyware, adware or using remote access to steal money or commit fraud; here the incentive is for profit.

Whatever the motivation malware writers have little or no regard for the consequences of their actions. Some malware has caused billions of dollars of damage, others have wasted years of work.

With the difficulty of detecting the origin of malware it is often impossible to apprehend and to prosecute the writer. Unfortunately it is relatively easy to develop malware with authoring programs available for download, and groups who cooperate and challenge each other to develop new strains.

To avoid the damage that a virus can cause users have to be both prudent and careful. The following strategies can be used to protect data:

- use anti-malware products such as virus scanners, spyware detectors and firewalls to protect the computer; these work by carefully monitoring the operations of the computer, if they detect an aberration they notify the user and, if directed, remove the malware
- avoid questionable applications where the originator is not known or trustworthy
- be extra careful in the use of portable media such as flash drives or external HDDs
- do not run executable email attachments or click on pop-ups without being very sure of their provenance
- since most malware operates in the Windows environment the use of a secure or non-Windows operating system such as Linux is very effective
- make regular backups of data in case of infection or data loss.

Unfortunately new malware strains are being developed all the time in an escalating battle with anti-malware companies. Most of these companies maintain a web site with up-to-date information on malware and from where up to date information can be obtained.

© Kevin Savage 2011 – Licensed to St Mary's Catholic College for use in 2012

## Activity 5.5 – Viruses

1. a What is malware?
   b Pick one of Trojan, zombie, or rootkit and explain why it is called that.
   c Find an example of where malware has caused damage, and briefly describe what happened and the effects the incident had.
2. a What is a *drive-by download*?
   b What forms of malware fit this category?
3. a What is a keylogger?
   b What might be some of the malicious purposes of installing a keylogger on a computer or system?
   c What forms of malware might install a keylogger on your computer?
4. a What is spam? Why is it used?
   b What is a denial-of-service (DoS) attack? What might be the motivations in carrying out a DoS attack?
   c How might malware be involved in setting up remote spam servers or carrying out a DoS attack?
5. Describe a good defensive strategy towards malware that you can take with your own home computer.
6. The news media often gives much prominence to the reporting of virus outbreaks.
   a Why does the media tend to feature these stories?
   b What is the image of viruses that the media tends to portray to the general public?
   c Suggest the potential effects of this form of portrayal.
   d There are many urban myths and spoofs that involve viruses. Find out and report on one virus myth or spoof and discuss its effect on users generally.
7. a What is meant by the term “social engineering”?
   b Why is a user more likely to open an email attachment when it apparently comes from someone they know? In your answer explain why such email viruses are able to spread so rapidly.
   c Find out what crimeware is. How is it distinct from spyware or adware? What social engineering aspects does crimeware utilise?
8. There is often an irrational fear of viruses based on limited knowledge. Sometimes viruses are blamed when there has only been a program fault. Why are people so ready to accept the presence of a virus or other malware in their computer?
9. Visit the web sites of anti-malware software companies and investigate the range of viruses and other malware around.

   Pick one and report on it.

© Kevin Savage 2011 – Licensed to St Mary's Catholic College for use in 2012

## Problem solving revisited

There are many different ways people have described systematic problem solving methods but the simplest and most elegant was developed by the Hungarian mathematician George Polya.

His method is:

1. Know what the problem is.
2. Plan a solution.
3. Try out the solution.
4. See how good the solution is.

George Polya

We do this all of the time without realising it – let's look at a problem to see how:

*Problem*: I'm hungry

*Step 1 – Know the problem*

- how hungry am I? (am I Big Mac hungry or just Snickers-bar hungry?)
- how much money do I have?
- where can I go to get un-hungry?

*Step 2 – Plan a solution*

- will I suffer in silence (not likely!)
- I'm really only peckish so I'll get a Snickers bar
- can I do some grovelling near my mates (no chance)
- I've got $2.50 so I have enough money
- I think I will head off to the shop

*Step 3 – Try out the solution*

- go to shop and buy Snickers bar
- eat it (yum!)

*Step 4 – See how good the solution is*

- not bad
- I've not got much money left now, but I'm not so hungry
- I'll probably do this the next time I'm peckish

As you can see the four steps are effective.

## Software Development Cycle

While Polya's problem solving process is effective for small problems it is not detailed enough for complex tasks such as a programming project. His method has therefore been expanded into what is known as the *Software Development Cycle* (SDC).

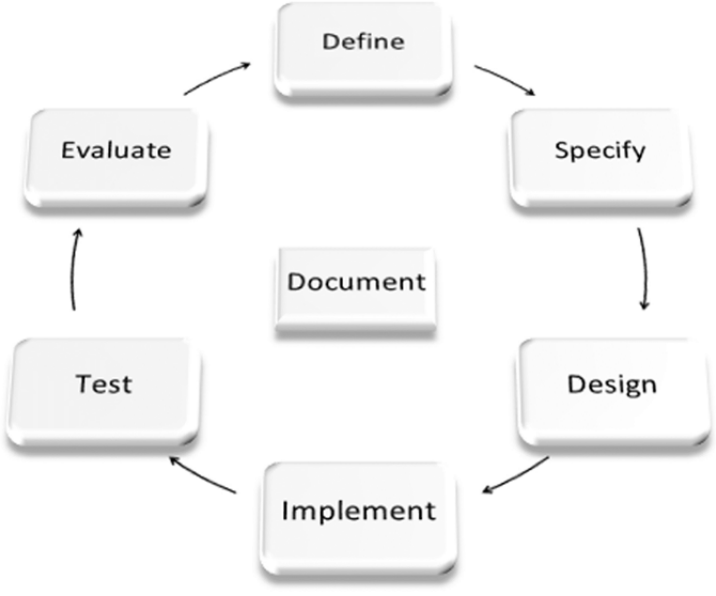
Shortly we will look at the SDC in action, but for now we will look at each of its parts in detail.

© Kevin Savage 2011 – Licensed to St Mary's Catholic College for use in 2012

### *Define*

The first step is to identify the problem to be solved.

We define the task to make sure we are very clear what we are about to do. This step is often left out, but not without wide reaching consequences. If we do not know the task we are about to complete, we will not even begin to know where to start. Added to this we could easily produce a beautiful solution, and then find it is of no use because it does not complete *this* task, but rather a similar one that we thought we were solving.




The Software Development Cycle

At this stage we should also not only decide what has to be done, but answer the question: *Who will use this program?* The level of expertise of the *end user* will strongly influence the user interface we create.

The definition can consist of an aim for the program, a general problem description, a rationale for development, and a list of assumptions and boundaries.

### *Specify*

The specification is a coherent conceptualisation (a clear mental picture) of what the program will do, without saying how this will be achieved. In the specification phase we do not look at how we will solve the problem, but rather find out where we will be when it is solved. Just what do we need to arrive at, at the end, to solve this problem?

The specification will involve a detailed description of how we expect the program will operate. To begin to accomplish this it is a good idea to state our objectives, list what we have been given, and what output the program will produce.

It is also important at the specification stage to again think of the end users of our program. We must consider how the running program will appear to the user and how they could be helped if

© Kevin Savage 2011 – Licensed to St Mary's Catholic College for use in 2012

in trouble. Making a program that works is important, but making a program that is *useable* is even more important.

If you have trouble developing the specification then cycle back to the definition phase as that has probably not been done properly and you do not really understand what you have to do.

### *Design*

The design phase is where we *map* the specification onto a *process*. This means that now that we are clear *what* we have to do, we start to think about *how* we will do it.

There are many possible ways of representing the solution. These include flow charts, Structured Development Charts, Nassi-Schneiderman diagrams, and pseudocode. You should select the method that suits you and the problem best.

In the design phase we must also be very aware of the HCI implications and make the program user friendly. It should be designed to be easy and pleasant to use, and to have help available where needed. Sketches of possible screen layouts will help with this.

If you are having trouble with the design phase it may be necessary to cycle back to the previous stage and ensure your specification is complete. (For example in the hungry problem, if we had not specified what sort of hungry, we might have sat down to a big meal when a Snickers bar would have been enough.)

### *Implement*

Implementation is where we put our design into place – we convert our plan into a computer program or programs.

The design should be flexible enough so that this could be done in any 3GL.

### *Test*

Throughout the implementation phase, and when it is completed the program must be tested.

A program that does not work is useless. A program that appears to work but has errors in it is even worse (why?). Good programmers thoroughly test each program to ensure it is error free. It is also important to make the program user-proof by preventing the user from getting into situations she or he cannot handle, or from entering impossible values.

To see if the program works we firstly try it with simple values to see if it gives predictable results. If all is right we then use a larger range of values to see if it still works. Eventually we try to "trick" the program to determine if it really is user-proof. This is called alpha testing.

For larger more complex programs, programmers use a set of *test data*, a wide range of values that will check as many possible variations on the program as possible. They will also get outsiders to test the application to ensure it really is user friendly and user-proof (*beta testing*).

### *Evaluate*

Evaluation is the carrying out of an appraisal of what has been done. It involves both ongoing evaluation and a formal evaluation at the end.

Ongoing evaluation has been in progress throughout the software development cycle. We make judgements as we go on aspects such as task requirements, screen presentation, user interface, algorithm design, implementation methods, and so on. At each stage we should also look at

© Kevin Savage 2011 – Licensed to St Mary's Catholic College for use in 2012

what we have done and decide if we are still on track. If not it is necessary to repeat the previous stage (cycle back).

Formal evaluation takes place when the project is finished. It involves looking back to see if the solution fits the specification. Have you done what you set out to do, and is it reliable, easy to use and free from errors? If you were a part of a team, how well did the group perform? You can also check to see if you have solved the problem the best possible way, to see if you have learnt anything you can use next time, or just to give yourself a pat on the back.

In some cases it is worth getting others to use your program so that you can get outside feedback, especially from someone who is not as familiar with it as you are.

### *Document*

Documentation is an ongoing activity throughout the software development cycle. At each stage you will need to record your progress.

In small programs documentation can be minimal, but in full-sized applications documentation is vital. Computer code can be extremely difficult to understand. It is part of your job as a programmer to ensure that other people (and even yourself) can understand what you have done in a program.

To do this you will be expected to make the program code itself understandable (internal documentation) and give some other written idea of what the program does (external documentation).

*Internal documentation* includes using meaningful identifiers for variables and subroutines, by spacing the program out with white space and indentation, and the use of effective comments.

*External documentation* will consist of your definition, program specification, the algorithms and pseudocode/flowcharts/NS diagrams, and the evaluation. It may also include a user manual or on-line help.

## Using the SDC

At first the SDC might appear cumbersome and time consuming, but it is a tried and tested method that is used widely in the software industry.

It is important to realise that the early steps in the software development cycle are the most important. The more time spent on the design stages, the less time there needs to be spent on the implementation, testing and possible redesign of the solution. It has been found in the computer industry that it is very expensive to alter a program that has been written if there are fundamental design errors, errors that may not show up until the program is in place and has been running for some time.

An indication of the importance given to the early stages of the SDC are the relative wages of software developers. Programmers, the people who actually *cut* (write) the code are the lowest paid. The systems analysts who design the specifications for the programs earn higher wages as their job is far more important in the long run.

The real value in programming is in the specification and design stages of the software development cycle. By using the SDC you will not only be using the method employed in the industry as the most effective, you will also be giving yourself every help in developing your own programs.

© Kevin Savage 2011 – Licensed to St Mary's Catholic College for use in 2012

## Activity 5.6 – SDC

1. a List the steps involved in the software development cycle?
   b Why is the SDC used in the software industry to prepare applications?
   c Why are the definition and specification steps so important?
2. Identify which step of the SDC each of the following might be:

   *Going on a date (not in order)*
   - I will ask dad for the car, pick up Sam, go to the movies, and return home
   - I think I will go with someone else next time
   - go on date
   - I will go with Sam to the movies
   - write about the date in my diary
   - tonight I want to go out with a friend
3. a What is the difference between programming and coding?
   b Which steps of the software development cycle correspond to programming, and which steps correspond to coding?
4. Which parts of the following correspond to the individual steps of the software development cycle?:

   Alan wants to build a wall to keep his dog in the back yard. He thinks about where he will have to put it, how high it will need to be, and what it will need to be made of. He also measures the place it is to go in. Once this is done he sits down and draws up a sketch of what it will be like, and gets a list of materials he will need. He then gets the materials from the hardware store and builds the wall. When he has finished he puts his dog in the yard to see if the wall keeps the dog in. As he worked he kept notes and sketches of what he was doing in case he needed to do this again. He was very pleased with the end result.
5. a What is the difference between internal and external documentation?
   b List the different forms of internal and external documentation.
   c Why is it important to document a program?

## The SDC in action

Now that we have seen the steps of the SDC we are now ready to use it to tackle a significant application in order to fully demonstrate how it can be used.

### Simulating an ATM

We are going to develop a simulation of an ATM (automatic teller machine) for the EastPac Banking Corporation. To achieve this we will follow the software development cycle and include an evaluation and full documentation.

Although this might not seem such a big deal to start with, after a little thought the problem might appear a little more daunting:

- where do we start?

© Kevin Savage 2011 – Licensed to St Mary's Catholic College for use in 2012

- how can the application be organised so that it does all that is required and does not leave anything out?
- what if we get half way through the solution and someone else has to finish it for us; can they take over without restarting?
- if we make errors in one part will it affect all of the rest?

These difficulties can occur with any large application and cause many problems for the software industry.

When a major software house decides to produce a new game or application they are faced with all of these obstacles and more. They have many constraints to satisfy, problems coordinating activities, difficulties with completeness, the need to document what is happening, and trouble maintaining a balance between many interrelated parts. With them it is made worse by the fact that applications are not written by an individual, but by teams of programmers.

Part of the solution is to utilise a process called *top down development*. This technique involves taking a large problem and dividing it into smaller parts called *modules*. In turn each of these modules is then subdivided into sub-modules until each is of a size that is easy to solve. Taken together the simpler sub-modules solve the whole complex problem.

Top down development, in turn, is a part of an approach called *structured programming***.** We will investigate both top down development and structured programming as we follow the software development cycle in the following task.

## ATM project – Define

To avoid wasted time working on the wrong task it is important to make a precise statement of what we intend to do. This will consist of an *aim* and a *general problem description*. If required we may include a *rationale for development* (reason for the program) and can list *assumptions* and *boundaries* of the task.

*Aim:* In this task we are going to develop a simulation of an ATM. The program is to be both user friendly and user proof with effective screen design and helpful prompts to the user. It will run with no errors and effectively simulate the workings of an ATM.

*General problem description:* The application will start by identifying itself and welcoming the user. The user will then have to enter a correct PIN to continue. If successful the user will be given options of account balance, deposit and withdrawal. These transactions will be simulated. On exit the application will thank the user. Account data will be stored to disc.

*Assumptions and boundaries:* As can be imagined with a simulation we will not actually be able to deposit or withdraw real money, but we can “go through the motions” of doing this. Also since we do not have, and cannot use, plastic cards we will work on an account number for each user. For simplicity we will limit the simulation to ten customers. While it is possible to read and write data to and from a random file in an ongoing manner, for ease of operation we will read all data into a record type at the beginning of the application, and then write it all back to disc on exit.

## ATM project – Specify

The specification is a detailed description, step by step, screen by screen, of what the program will do. At this stage we do not yet worry about how we intend to solve the problem, but instead we decide what the program will be like when it is completed. When we have finished developing the application we will compare the completed program with the specification to see if it has met these requirements.

© Kevin Savage 2011 – Licensed to St Mary's Catholic College for use in 2012

In the industry the specification for a program can be legally binding. If a programmer or company contracts to develop an application, they are then bound by the detail of the agreed specification. If the application produced does not meet this specification then the contract has not been fulfilled and compensation may need to be made.

To develop the specification the following process is helpful:

- prepare a list of inputs to the program (including disc input if required)
- identify outputs from the program (including to disc)
- determine what calculations must be performed (at this stage do not yet worry how these will be done).

For this task:

> *Inputs: user's account number and PIN, button clicks to move from screen to screen, banking option chosen, money amount to deposit or withdraw, account information read from disc.*
>
> *Outputs: greeting and farewell to user, instructions and prompts for the user, account balance, data written back to disc.*
>
> *Processing: comparison of account number and PIN with data read from disc, checking valid amount to withdraw, calculation of new balance after deposit or withdrawal.*

From this we can work out the sequence of steps that will be required to complete the above. This can be done by numbering them in the order they will occur in the program. When this has been done make decisions about program presentation, appearance, and user assistance.

Finally combine all of the above in a coherent written description that will list the objectives of the program, a step by step sequence of what will occur in the program, an overview of the user interface, and an indication of how the user will be given assistance.

> *Objective: The objective is to develop a working simulation of an ATM. To achieve this account data must be read from and later written back to disc, user identity will need to be checked, and account balances displayed or updated. A pleasant, helpful and effective interface will be provided for the user.*
>
> *Specification: The simulation will start with a splash screen to welcome the user. The program will then proceed to a main screen where transactions will occur. A file of customer information (account number, PIN and balance) for ten customers will be read from disc as this happens.*
>
> *For verification the user will be prompted to enter their account number and PIN. This will take place in a dialog box on top of the transaction screen. The program will check if the PIN is correct for that account number. If the PIN is wrong the user will be permitted two further attempts to get it right before the program terminates.*
>
> *If the PIN is correct the user will be given access to the transaction screen. This will give options of account balance, deposit, or withdrawal from the account. It will also have an exit option.*
>
> *Balance will be displayed on screen. Deposit will prompt the user for the amount and increment the account by that amount. Withdrawals will be limited to $500, and will not be permitted for amounts greater than the account balance. The account will be decremented by the stated amount. Negative values will not be accepted for deposit or withdrawal.*
>
> *On exit the program will write updated information back to disc and will display a closing screen.*
>
> *Tool tips and on-screen prompts will assist users of the ATM. There is no need for a help file or user manual for such a straightforward application. The program will have detailed internal documentation.*

© Kevin Savage 2011 – Licensed to St Mary's Catholic College for use in 2012

## ATM project – Design

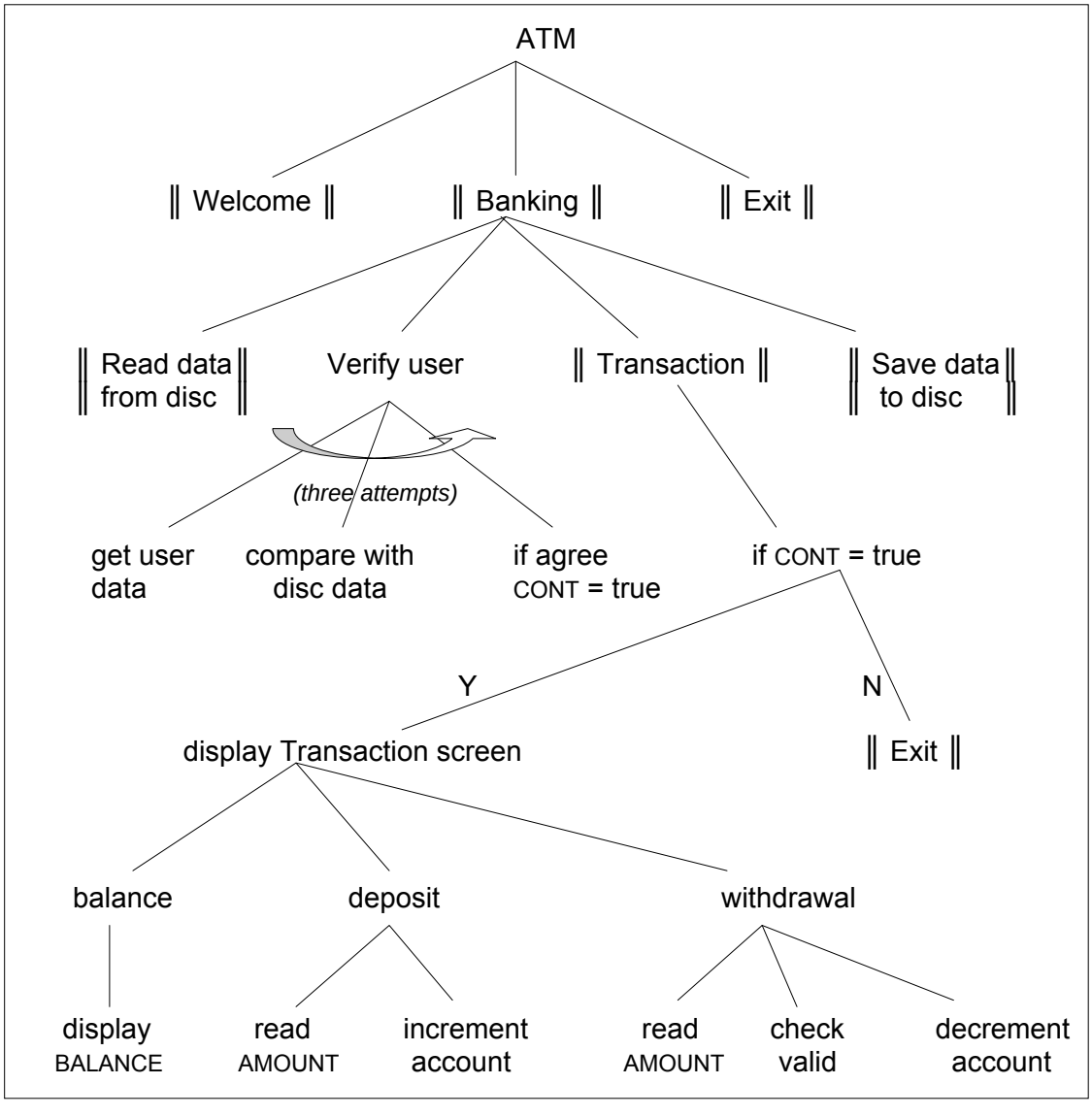
Now that we have a clear idea of what we want to do it is time to start designing the solution.

The design phase will consist of a structure chart, pseudocode, and screen designs.

### *Structure chart*

Since this is a much bigger task than we have attempted before we will use the top down method to divide it into modules and sub-modules based on our specification. By using this "divide and conquer" approach the problem can be tackled in sections.

This process can best be demonstrated using a *structure chart*:




© Kevin Savage 2011 – Licensed to St Mary's Catholic College for use in 2012

The simulation has been split into three modules – *Welcome* will greet the user, *Banking* is the using of the ATM, and *Exit* is the closing screen. In turn banking has been split into four sub-modules. In each case the ‖ indicates a module.

Now we look at each module in turn and see if we can easily construct what happens. If we cannot then we must sub-divide it further. When each step is simple enough we can represent it in pseudocode.

### *Pseudocode*

The *Welcome* and *Exit* modules are relatively simple to prepare. Each can easily be put together from screen design sketches. Since these modules are at a level that is easy to prepare we will leave them and work on the *Banking* module.

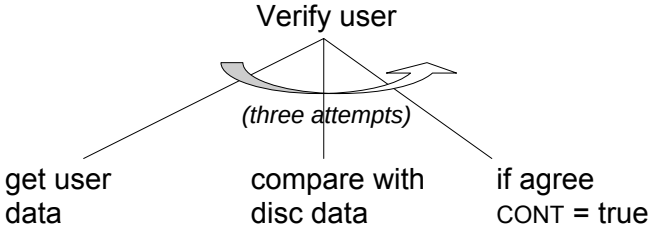
The reading and writing data to and from disc modules can be represented in pseudocode in the way done earlier in the section on file handling. The modules for *read data from disc* and *save data to disc* are similar and can be written in pseudocode as:

```
Read
    reset custData
    count = 0
    while not eof(custData)
        count = count + 1
        read(custData, customer[count])
    endwhile
```

```
Write
    rewrite custData
    do acct = 1 to count
        write(custData, customer [acct]
    enddo
    close custData
```

Customer information will have been saved in a file that we will create separately. This file will hold the account number, PIN, and balance for each of our ten customers (i.e. thirty pieces of data). Each piece of data is read into the record type array *customer*.

*Verify user* consists of comparing the customer number and PIN with the data read from disc. The user is to be given three attempts to remember their PIN:




The curved arrow indicates the parts of the program to be repeated as we have to give the user three attempts to remember their PIN. For each attempt we will compare the values they enter with each set of account data that has been read into the *customer* record:

```
Verify
    go = 0
    cont = false
    while go < 3 and cont = false
        go = go + 1
        read accTry
```

© Kevin Savage 2011 – Licensed to St Mary's Catholic College for use in 2012

```
        read pinTry
    endwhile
    do acct = 1 to 10
        if customer(acct).number = accTry and customer(acct).pin = pinTry then
            cont = true
            cust = acct
        endif
    enddo
```

The variables *cont* and *go* are used to control access to the transactions. *Cont* only becomes true if the account number and PIN the user enters agree with the values read from disc. *Go* is used to give the user three attempts.

The *do* loop works by comparing the account number and the PIN the user entered (*accTry* and *pinTry*) with the values in the *Customer* record. If a match is found *cont* becomes true and the account that was found (*cust*) is stored for use in transactions.

At this level the pseudocode should be easy to write in a 3GL and so we move to the transaction module.

Provided a correct PIN is entered (*cont* is true) the transaction screen is displayed and the user can then make a transaction; if not the program will exit.

If the user chooses balance all that is required is a simple display of the money currently in their account. After the display control is to return to the transaction module.

The pseudocode for deposit and withdrawal are relatively simple:

```
Deposit
    read amount
    if amount > 0 then
        balance = balance + amount
    endif
```

The *if amount > 0* line is to ensure the user does not enter a negative amount of money.

*Withdrawal* must make two other checks:

```
Withdrawal
    read amount
    if (amount < balance) and (amount < 500) and (amount > 0) then
        balance = balance – amount
    endif
```

These are examples of user proofing the program (i.e. not letting the user interfere with the correct running of the application). If the user does enter an invalid amount of money however, at the moment the user is not given another chance. Perhaps we could improve the algorithms to read:

```
Deposit
    amount = -1
    while amount < 0
        read amount
```

© Kevin Savage 2011 – Licensed to St Mary's Catholic College for use in 2012

```
        endwhile
        balance = balance + amount
    Withdrawal
        amount = -1
        while (amount > 500) and (amount < 0)
            read amount
            if amount > balance then
                write “Error – withdrawal amount greater than balance”
            else
                balance = balance – amount
            endif
        endwhile
```

These will each keep in the *while* loop until the user enters a valid value.

### *Screen designs*

With the structure chart and algorithms developed, all that is left to do as part of the design phase is to prepare sketches to indicate the proposed layout of the screens.

In doing this we must be very aware of the HCI aspects. The screens should be well laid out and easy to use.

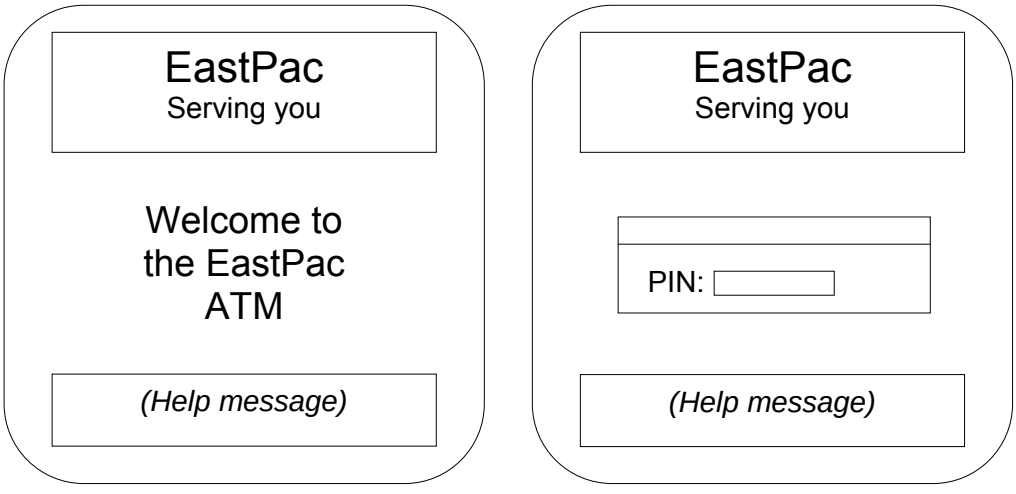
The following seven principles have been refined by Microsoft after extensive research into what makes a good interface design.

1. The user is in control – the user initiates actions, not the computer.
2. Directness – users should see how their actions are being carried out, e.g. drag and drop, GUI.
3. Consistency – consistency in naming conventions, visual presentation, operational behaviour and in placement makes for a more stable, predictable environment for the user and allows for the transfer of concepts more easily.
4. Forgiveness – programs should be fault, error and mistake tolerant; allow users to learn by trial and error (this is also known as *interaction discovery*).
5. Feedback – confirm the program is doing what the user expects.
6. Aesthetics – maintain a simple coherent environment that communicates important cues to the user.
7. Simplicity – a simple (not simplistic) environment is easy to learn and use; to do this use familiar concepts or metaphors, limit verbiage (words), and balance the need for functionality with the need for simplicity; use progressive disclosure by showing information only when it is needed or appropriate e.g. menus, successive dialogue boxes.

Look to use these principles as a guide in your interface development.

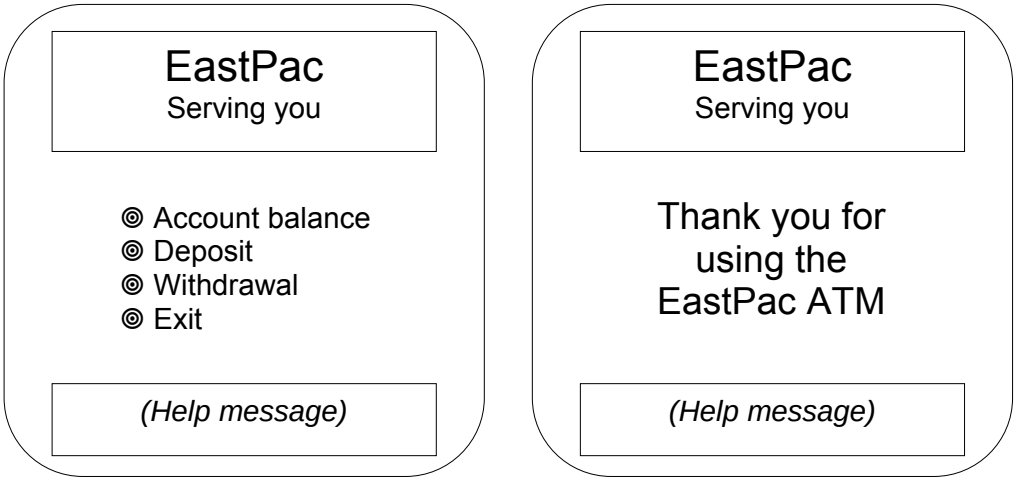
© Kevin Savage 2011 – Licensed to St Mary's Catholic College for use in 2012

Screen designs:




Welcome screen

Verification screen




Transaction screen

Exit screen

Before we get to the implementation stage note that if we have completed the first three steps of the software development cycle properly we should be able to apply our design to *any* 3GL. Remember it is the design stages that are important in developing an application. Once the design is complete the remainder (coding and testing) is largely a mechanical task.

## Structured programming

We have used top down development to cover the first three steps of the software development cycle.

In summary the process is:

- describe the problem in your own words to find out what it entails

© Kevin Savage 2011 – Licensed to St Mary's Catholic College for use in 2012

- prepare a specification of what the program should actually do
- break the problem into parts (do not concern yourself yet with how each of the parts will be solved)
- take each of the modules at the next lower level in turn and repeat the process of subdivision until you arrive at a level where each sub-module itself is easy to code
- use structure charts and pseudocode (or similar) to represent the process.

It is important that all of the sub-modules of a particular module are developed before going to the next lower level. At each level check that the sub-modules, taken together, will completely satisfy the module they came from. This is necessary to ensure that the modules will work together before getting into the more complex details at the lower levels. We will be using a method called *module testing* to check this.

Top down development is part of a strategy known as structured programming that is used extensively in the computer industry (and beyond). This approach requires that a program be written in such a way that it can be subdivided into clearly distinguishable modules that are fully documented, each of which can be treated as a self contained unit. It is necessary to make sure all modules fit together correctly and communicate with each other to meet the specifications, that the program is well documented to improve coordination throughout the programming team, and that testing and debugging are made simple.

The advantages of structured programming are:

- ease of design – the details do not obscure the task
- understanding – each level of modularisation is simple enough to comprehend in itself without reference to the rest of the process
- independence – separate modules can be written by different members of a team
- portability – modules written for one program can be used in different applications
- isolated debugging – an error in one module can be located and corrected without having to refer to other modules
- flexibility – if the requirements for the program change, just the affected modules need to be rewritten
- documentation – all aspects are carefully recorded and easily communicated.

Structured Programming relies on careful planning, good coordination, and detailed documentation. It is the basis of effective programming in the software industry and you are expected to use its methods in developing your own programs.

In 3GLs we have two options to make our programs modular.

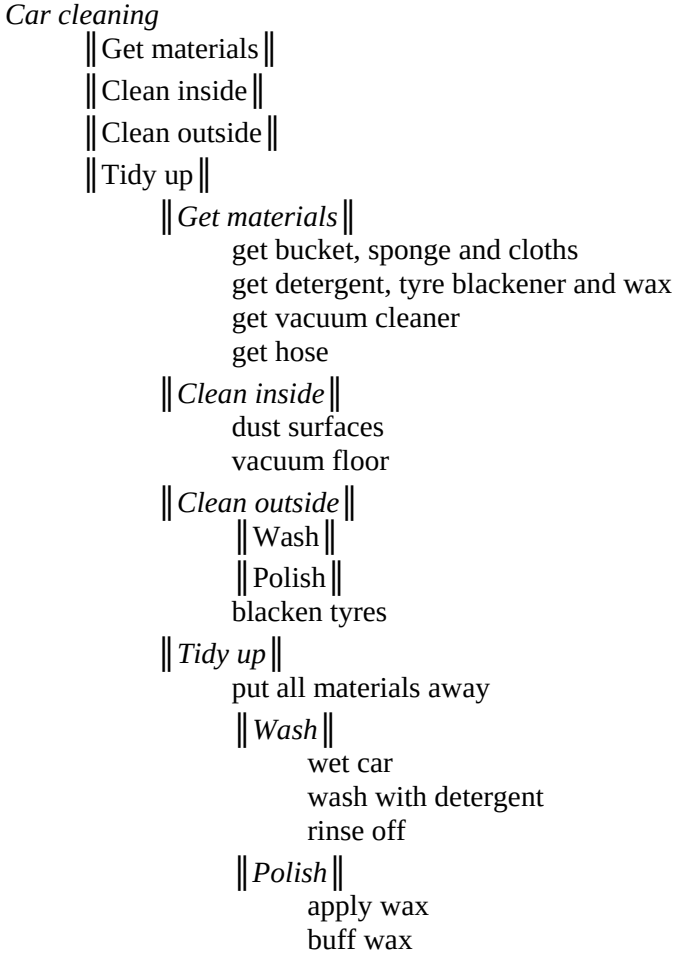
The first method is to divide the application into separate units. In this approach each unit has a screen or screens and computer code associated with it. These units can be treated as modules, each with all of the advantages listed above (independent, portable, etc.). This approach is useful in very large applications and is generally used by teams of programmers.

A second simpler approach is to create the application with subroutines and blocks of code to represent the structure chart modules.

© Kevin Savage 2011 – Licensed to St Mary's Catholic College for use in 2012

## Activity 5.7 – Structure

1. Mike has decided to clean his new car. Having never done this before his friend has written the tasks to be done onto a sheet of paper. Present these as a top down structure chart:




2. Your sister is going to get married soon and she has asked you to take care of the reception. Without thinking you have foolishly agreed.

   Use the top down development method to solve this problem. Start by defining the problem and then specify what the solution will entail (just what you going to have to do). Then, working from this, break the problem into parts, and then break these parts down once again.

   Represent your solution as a structure chart.

3. Bert Mudge is part of the team developing *Chaos* a new FPS game on the X-box. Over the past three months he has been working on the representation of explosions on the screen. It

© Kevin Savage 2011 – Licensed to St Mary's Catholic College for use in 2012

is his job to ensure that graphics, sound and impacts are coordinated. Last week he was offered a big rise in salary by Sony and so has packed his bags and gone.

Briefly discuss the effects of the above situation on two software houses, the first of which uses the structured approach, and the second does not.

4. We are now ready to code and test our design. This is the implementation stage of the SDC for the ATM.

   Because this is such a large undertaking we will conduct it in three phases, create the data file, make and link the application screens, and finally generate the computer code from the algorithms.

   a Before you begin you will need a *cust.dat* file on disc to hold information on customer account numbers, PINs and account balances. To do this write a separate small application to do the job for you.

   Establish *account* as a record with fields for account number, PIN and account balance. *custData* is the disc file and *customer* is an array for 10 accounts with the same fields. The information is to be read from the *customer* array onto the *custData* disc file.

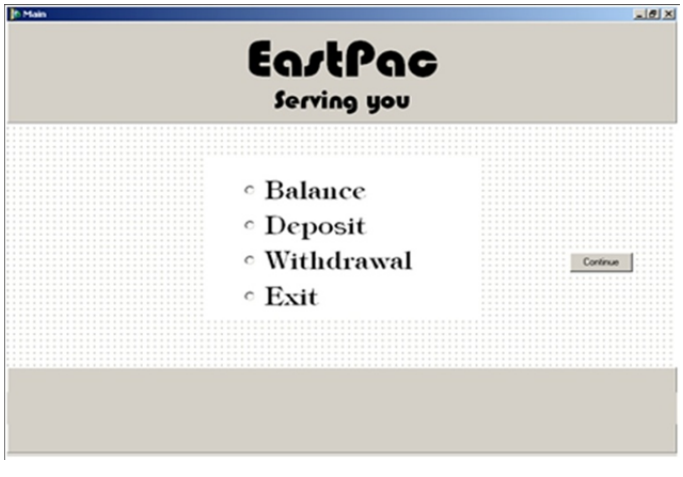
   When prompted enter the account number, PIN and account balance (e.g. 3412, 7788, 234.75) for ten customers. (Make a note of the account number and PIN of several of these as you will need them later to test your program.)

   b Create screens for the ATM.

   To do this prepare a basic layout with the bank logo that you will use for the application. Use this as a template (or basic outline) for all three screens you will use, i.e. Welcome, Main and Farewell.

   Welcome ⇨ Main ⇨ Exit

   These will form the basis of our welcome, banking and exit screens from the top level of our structure chart.




   Develop these screens and write code to link them, one to the next.

© Kevin Savage 2011 – Licensed to St Mary's Catholic College for use in 2012

Run to test. While nothing will happen other than the screens appearing, at least we can check that to this stage the application is working correctly in sequence. Once we are sure this level is correct then we can get onto developing the code to run the application.

c Write the program to run the application:

i Prepare the code to read the data in *cust.dat* into the *customer* array as the application starts. When the program runs there should be ten sets of data (account number, PIN, and account balance) in *customer*.

ii Next enter the verify procedure to check the user can match an account number to a PIN.

iii Code the transaction module with subroutines for balance, deposit, and withdrawal. This module should only be run if *cont* is *true* (i.e. the user has entered a valid account number and PIN).

iv Save, run and test to see the balance is correctly displayed, that the customer can deposit or withdraw and that the new balances are saved to disc.

v Link and test all parts of the program.

## Module testing

We have now completed the implementation phase of the ATM project. As we went we have used a process called module testing to apply our plan.

*Module testing* is a way of implementing a program so that errors are isolated easily. It involves coding each part of the program section by section, and testing each part at each step. If there had been an error, it could only have occurred in the section just written.

We first created our three screens and then, without writing the rest of the program, we ran the simulation to see if the three screens appeared in order. If there was a mistake we could easily locate it. We then implemented the read from disc procedure and tested it, then the verify procedure and tested it, and so on.

Rather than having to look through a whole completed program for errors, we narrowed down where they could occur. This makes them easier to detect and remove. Once we know a section is correct we can confidently move onto the next module.

## ATM project – Test

We are now up to the next step of the software development cycle, testing. Actually we have been using module testing as we have been going to see if it compiles properly, but now we must test it more rigorously.

Run your program again and see if you can “trip it up”. Is it user proof? Can the user enter invalid values? Can you cause it to “crash” by pressing the wrong button? (If so can you fix the problem?)

After testing by yourself the program should be beta tested by a user who has not seen it before. The original programmer by this stage is so familiar with the program that they will often skip over errors, or not see faults that a new user would. Comments from an outside user, not only

© Kevin Savage 2011 – Licensed to St Mary's Catholic College for use in 2012

on performance and operation, but also on appearance and useability can greatly improve an application.

## ATM project – Evaluate

At this stage although we have spent a long time developing the application we have not finished yet. It is necessary to evaluate how well we have done. We check to see if the program has met specifications and make a note of things we have learned so that we will not make the same mistakes next time.

It is also important to let potential users of the program try it to see if it does what they want, and is user friendly and user proof. Comments can also be made on performance, reliability and possible improvements.

Evaluation should not only be of the product (the application) but also the process (how the application was developed). The following is an indication of how this might be done.

Evaluation: The ATM simulation works well. It can start as a *Windows* executable file from the desktop and runs to completion smoothly.

The application meets the basic specifications of splash screen, verification (giving the user three attempts to enter the correct PIN), and transactions (updating account balance). Data is read from and written to disc effectively and running account balances thus maintained.

The program does not permit the user to enter invalid values, however the program is not user proof. If the user enters non-numeric data for account number, PIN, or amounts to deposit or withdraw the program will crash. To correct this the code will need to be amended to only accept numeric data at these points. The program is user friendly with help messages at all stages. It is well presented with clear, easy to see buttons and labels.

Potential users have tried the program and found it easy to use, effective and reliable. These users however pointed out two misconceptions. Firstly while an ATM can accept odd amounts of money for deposit (e.g. cheque amounts) an ATM should only dispense money in multiples of $20 and $50. The second inaccuracy was that banks do not let customers empty their accounts, usually expecting them to leave at least $1 in the account to keep it open. Apart from this the simulation was felt to be realistic as far as it went.

Besides the problems mentioned above, other improvements could be made to make the simulation more life like. Perhaps we could include a print out of transactions, or even include graphics to show money coming out of a slot in the machine. Presenting the user with a key pad to enter values would make the simulation more realistic.

Finally when we enter the PIN it would be good if asterisks appeared on screen instead of the PIN itself. This can be done by including a password dialog as part of the simulation.

Development proceeded smoothly with few problems encountered along the way. Since the program was developed with guidance there were no problems in coordinating activities.

On the whole a workable realistic simulation with a few minor faults that can be rectified.

## ATM project – Document

Although documentation is listed at this stage of the software development cycle, it is an ongoing part of the process. It needs to be collected at every step and compiled at the end.

Documentation records everything we have done so that we, or others, can reference what decisions we made and gain some understanding of how the program works.

For this project our documentation will include:

- problem definition

© Kevin Savage 2011 – Licensed to St Mary's Catholic College for use in 2012

- solution specification
- screen designs
- structure charts
- pseudocode and algorithms
- a print out of the code including clear internal documentation
- a copy of the files on disc
- evaluation (including user evaluation)
- user manual (if it is felt one is needed).

## Activity 5.8 – ATM follow up

Make improvements to the ATM simulation as suggested in the evaluation above.

1. a Amend the withdrawal procedure so that the customer cannot completely empty the account.

   b Alter the withdrawal procedure so that the ATM will only dispense money in multiples of $20 and $50.

2. Add a number keypad so that the user can only enter number values.

3. Investigate using a sub-screen of your own to accept the user's PIN.

   Set this up to only display asterisks as the user types, and limit the display so only a four digit PIN can be entered.

4. Investigate using graphics to simulate money coming out of the ATM.

5. Investigate *printing* to provide a printed receipt of transactions.

## Possible programming tasks

The following are ideas for suitable projects.

1. A program that will handle the bookings for your school musical. The program will need to record names and contact phone numbers, the number of seats and the night booked. It should be capable of permitting cancellations or alterations to bookings.

2. A poker machine simulation that includes suitable graphics of playing cards for display. The amount spent will be updated and the user has the opportunity to win jackpots

3. A virtual pet that must be fed and looked after or it will die. Suitable graphics, realistic "life" values, and effective user feedback will greatly enhance this application.

4. A TV quiz show such as *Who Wants To Be A Millionaire?* or *Wheel of Fortune*.

5. A life cycle program (like *The Sims*) for a village, livestock or a national economy whereby the user makes decisions from generation to generation to aim at some goal.

© Kevin Savage 2011 – Licensed to St Mary's Catholic College for use in 2012

6. A program that simulates a race between horses or cars with a random function to determine movement. Users can predict the winner, and "bet" on which will win. Odds that affect the result will enhance the program. A record of money won or lost is to be kept.

7. A music program that draws music score sheets and gives the user a selection of notes they can use. The user can then compose and play a song.

8. A costing program for one of the following:
   - a kitchen – the user enters the dimensions of cabinets, what finish, bench tops and doors wanted; the program calculates the total cost of the conversion
   - a computer store – the user identifies components in the computer they want and the program calculates the total cost
   - a fast food restaurant.

10. A program that presents a multiple choice test that could be for:
    - a driver's license - this could include graphics of intersections, etc.
    - a primary school maths or spelling test.

11. A tic tac toe game (nought and crosses).

12. A biorhythms program where the user enters their date of birth and the program plots their current biorhythms for intellectual, physical and emotional levels.

13. A simulation to show how gas will diffuse from one container connected to another. This program will start with 100 molecules of gas in one container, and then using a random function show how some of these molecules will diffuse into the other container. As diffusion continues some molecules from the second container will drift back into the original. The program should provide a count of the number of molecules in each container and have a graphical representation of what is happening.

14. An address book to record the names, ages, addresses and phone numbers of friends. The address book must include the ability to add new names, update information and, if necessary, delete people who have moved away.

15. A snack machine simulation based on the following scenario:

    Gobble'n'Go are about to install snack machines in business offices. Each snack machine can deliver a small range of confectionary and soft drinks. The various chocolate bars etc. are individually priced, while all drinks are a set price depending on size, small, medium or large. Each machine has access to a customer data file for the given workplace. The file consists of a list of customer personal identification numbers (PINs) and amount owing by that customer (initially $0.00).

    In use, a customer must enter a valid user PIN before being permitted to place an order. Once the order is placed it is displayed on a screen with its total cost, and confirmation is requested. The program must keep a running total of amounts owed by each customer until exit, at which time the data is written back to disc. No customer is permitted to run up a bill of over $30.00.

© Kevin Savage 2011 – Licensed to St Mary's Catholic College for use in 2012